Soluzione Esercizi di Programmazione 1

Ivano Salvo
Università di Roma "La Sapienza"
email: salvo@dsi.uniroma1.it

Anno Accademico 2003-04

Esercizio 1 *Scrivere una funzione iterativa ed una ricorsiva che calcolino l'esponenziale sfruttando le seguenti uguaglianze:*

$$\begin{aligned} m^{2n} &= (m \times m)^n \\ m^{2n+1} &= m \times m^{2n} \\ m^0 &= 1 \end{aligned}$$

Soluzione Osserviamo, per i non esperti di definizioni induttive, che la defizione di esponenziale data sopra è data per induzioni sull'esponente. Viene definito il significato dell'esponenziale nel caso in cui l'esponente sia un numero pari (maggiore di zero!), un numero dispari oppure zero. In ogni caso il significato viene dato induttivamente in termini di un esponenziale con esponente minore.

La funzione ricorsiva ispirata alle equazioni ricorsive sopra scritte si ottiene semplicemente traducendo opportunamente in C le equazioni matematiche:

```
int espRec(int m, int n)
{ if (n==0) return 1;
     else if (n % 2 == 0) return espRec(m*m, n / 2);
     else return m*espRec(m, n-1);
}
```

Osserviamo, senza entrare nei dettagli, che questo programma fa un numero *logaritmico* di moltiplicazioni. Osserviamo anche che il classico algoritmo che esegue un numero *lineare* di moltiplicazioni:

```
int espRec(int m, int n)
{ if (n==0) return 1;
     else return m*espRec(m, n-1);
}
```

corrisponde alla seguente definizione induttiva di esponenziale:

$$\begin{aligned} m^{n+1} &= m \times m^n \\ m^0 &= 1 \end{aligned}$$

La versione iterativa di questa algoritmo dovrebbe essere ben nota al lettore:

```
int espIt(int m, int n)
{ int i;
  int esp = 1;

  for (i=1; i<=n; i++) {esp = esp*m;}
  return esp;
}
```

La versione iterativa richiesta, invece deve tener conto di trattare in modo diverso esponenti pari e dispari.

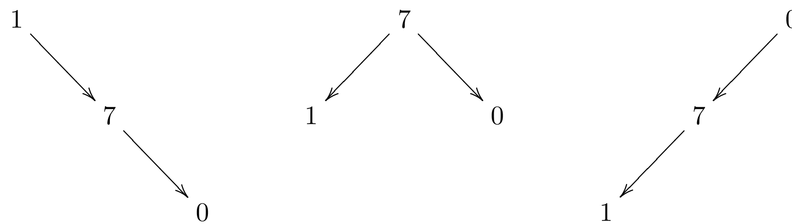
```
int espIter(int m, int n)
{ int esp=1;

  while (n!=0)
    { if (n % 2 == 0)
        { m = m*m;
          n = n/2;}
      else
     { esp = esp*m;
       n--;}
  }
  return esp;
}
```

Esercizio 2 *Scrivere una funzione:*

`Tree makeTree (int visita[], int inf, int sup)`

che costruisce un albero binario che ha come visita inorder (o simmetrica) la sequenza memorizzata nel vettore.

Soluzione Osserviamo che l'esercizio richiede di costruire *un qualsiasi* albero che abbia come visita la sequenza memorizzata nel vettore. Quindi dato un vettore contenente la sequenza `1 7 0`, la funzione rispetta la specifica richiesta se fornisce come risultato uno qualsiasi dei seguenti alberi:



(In realtà anche altri! Chi li vede?) L'idea è molto semplice. La funzione `maketree` sceglie in modo arbitrario un elemento della sequenza compresa tra gli indici `inf` e `sup` come radice dell'albero. Alloca memoria per creare un nuovo nodo dell'albero. Dopodichè chiama ricorsivamente se stessa, costruendo il sottoalbero sinistro a partire dalla sottosequenza che sta *a sinistra* dell'elemento scelto come radice, e il sottoalbero destro a partire dalla

sottosequenza che sta *a destra*. La sottosequenza su cui lavora una particolare attivazione della funzione `maketree`, è determinata dai parametri ausiliari `inf` e `sup`. La ricorsione terminerà quando la sequenza è vuota, cioè quando `sup` è minore di `inf`. La sequenza vuota è chiaramente una visita inorder dell'albero vuoto. La prima chiamata avrà quindi la forma:

```
Tree T = makeTree (v, 0, N-1)
```

dove `N` è una variabile che mantiene la lunghezza del vettore `v`. A questo punto, possiamo sbizzarrirci a creare gli alberi delle forme, più strane. Ad esempio, possiamo decidere di scrivere la funzione che ricostruisce l'albero tutto sbilanciato a destra:

```
tree makeTreeRight(int visita[], int inf, int sup)
{ tree temp;

  if (inf > sup) return NULL;
  temp = (tree) malloc(sizeof(treeNode));
  temp->info = visita[inf];
  temp->left = NULL;
  temp->right = makeTreeRight(visita, inf+1, sup);
  return temp;
}
```

o quello tutto sbilanciato a sinistra:

```
tree makeTreeLeft(int visita[], int inf, int sup)
{ tree temp;

  if (inf > sup) return NULL;
  temp = (tree) malloc(sizeof(treeNode));
  temp->info = visita[sup];
  temp->left = makeTreeLeft(visita, inf, sup-1);
  temp->right = NULL;
  return temp;
}
```

o quello bilanciato:

```
tree makeTreeBalanced(int visita[], int inf, int sup)
{ int medio;
  tree temp;

  if (inf > sup) return NULL;
  medio = (inf + sup) / 2;
  temp = (tree) malloc(sizeof(treeNode));
  temp->info = visita[medio];
  temp->left = makeTreeBalanced(visita, inf, medio-1);
  temp->right = makeTreeBalanced(visita, medio+1, sup);
  return temp;
}
```

o uno in cui la scelta della radice è casuale:

```
tree makeTreeRandom(int visita[], int inf, int sup)
{ tree temp;
  int r;

  if (inf > sup) return NULL;
  if (inf == sup) r = inf;
      else r = rand() % (sup - inf) + inf;
  temp = (tree) malloc(sizeof(treeNode));
  temp->info = visita[r];
  temp->left = makeTreeRandom(visita, inf, r-1);
  temp->right = makeTreeRandom(visita, r+1, sup);
  return temp;
}
```

in tutti i casi, è importante che gli elementi che andranno nel sottoalbero sinistro (rispett. destro) stiano nel vettore a sinistra (rispett. destra) della radice scelta.

Esercizio 3 *Scrivere una funzione RICORSIVA che fornisca il massimo elemento di una matrice di interi di dimensione* $n \times m$.
Soluzione E' sufficiente escogitare un modo per scorrere la matrice in modo ricorsivo. Probabilmente il lettore non ha difficoltà scrivere la funzione iterativa:

```
int maxMatrice(int M[][H], int n, int m)
{  int i, j, max;

   max = M[0][0];
   for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        if (max<M[i][j]) max=M[i][j];
   return max;
}
```

Si tratta di individuare un modo per generare tutte le coppie di indici, in modo ricorsivo. Un'idea banale è quella di cominciare con gli indici riga e colonna entrambi a 0. Dopodichè ad ogni chiamata ricorsiva si incrementa l'indice riga fino a quando raggiunge il numero di colonne. A quel punto è necessario riazzerare l'indice riga e incrementare l'indice colonna di 1. I parametri `n` ed `m` sono rispettivamente il numero di righe e di colonne. La prima chiamata è del tipo:

```
max = maxMatRec(M, n, m, 0, 0)
```

.
La ricorsione termina quando l'indice riga raggiunge `n-1`, e l'indice colonna raggiunge `m-1`. In tal caso il massimo della matrice è proprio `M[n-1][m-1]`.

```
int maxMatRec(int M[][H], int n, int m, int i, int j)
{ int x,y;

  if (i==n-1 && j==m-1) return M[i][j];
  if (i==n-1) { x=0; y=j+1;}
         else { x=i+1; y=j;}
  return max(M[i][j], maxMatRec(M, n, m, x, y));
}
```

Per coloro che non sanno resistere al fascino di scrivere meno righe di codice possibile e che non amano le variabili inutili, diamo anche la seguente soluzione:

```
int maxMatRec2(int M[][H], int n, int m, int i, int j)
{ if (i==n-1 && j==m-1) return M[i][j];
  return max(M[i][j], maxMatRec2(M, n, m, (i+1)%n, j+(i+1)/n));
}
```

Osservate che l'algoritmo ricorsivo e quello iterativo si comportano in modo un po' diverso: quello iterativo calcola il massimo scorrendo "in avanti" la matrice, mentre quello ricorsivo lo calcola "al rientro" dalle chiamate ricorsive, quindi, all'indietro. E' possibile anche scrivere un programma ricorsivo che si comporta come quello iterativo, al prezzo di usare un parametro ausiliario:

```
int maxMatRec(int M[][H], int n, int m, int i, int j, int max)
{ int x,y;

  if (i==n-1 && j==m-1) return max;
  if (i==n-1) { x=0; y=j+1;}
         else { x=i+1; y=j;}
  if (max < M[i][j]) max = M[i][j];
  return maxMatRec(M, n, m, x, y, max);
}
```

In questo caso una possibile chiamata iniziale è la seguente:

```
max = maxMatRec(M, n, m, 1, 0, M[0][0])
```

.

Esercizio 4 *Scrivere una funzione che, data una lista L di interi ed un intero k, restituisca TRUE se k si può ottenere giustapponendo due o più elementi consecutivi di L, FALSE altrimenti. Esempio: sia*

$$L = 2 \to 36 \to 4 \to 2 \to 17 \to 88.$$

Se $k = 4217$ oppure $k = 364$, la funziione deve restituire TRUE; se invece $k = 24$ oppure $k = 230$, la funzione deve restituire FALSE.

Soluzione Si tratta di un esercizio obiettivamente complicato, pieno di insidie. Cerchiamo di isolare passi più semplici. Innanzitutto se la lista è vuota, non ho più speranza di trovare l'intero k. Un altro caso facile si verifica quando la lista contiene l'elemento k. In questo

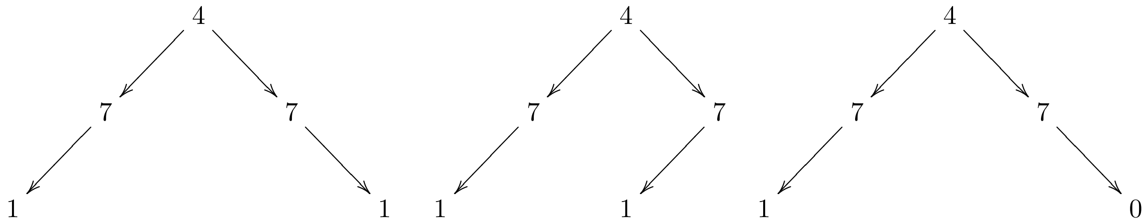
caso torno subito 1 al chiamante. Il caso un po' più complicato si ha quando è necessario verificare se l'elemento della lista sia uguale *a una sequenza iniziale* di k. Questo si fa semplicemente verificando TUTTE le sequenze iniziali di k, ottenendole dividendo via via k per 10. Ad esempio per verificare se 4 è una sequenza iniziale di 4217, verifico se 4 sia uguale a 4217, 421, 42 e 4. A questo punto mi sono ricondotto al problema originale, e devo cercare il numero 217 nel resto della lista, con due accortezze:

1. affinchè 4217 si ottenga per giustapposizione di elementi della lista, è necessario che 217 si ottenga giustapponendo elementi di *di una sequenza iniziale* del resto della lista. Per questo bisogna scrivere una funzione leggermente diversa, `ricercaInizio`.

2. Se 217 non fosse la sequenza iniziale del resto della lista, non posso concludere che 4217 non sia presente. Infatti ci potrebbe essere un altro elemento uguale a 4 o 42 etc. più avanti (pensate ad esempio alla lista $4 \to 36 \to 4 \to 2 \to 17$);

```
int ricerca(lista L, int k)
{ int j=1;
  int kaux = k;

  if (!L) return 0;

  if (L->elem == k) return 1;

  while(kaux > L->elem)
{ kaux = kaux / 10;
          j = j*10;
          }
  if (kaux == L->elem)
     if (ricercaInizio(L->next, k % j)) return 1;
  return ricerca(L->next, k);
}

int ricercaInizio(lista L, int k)
{ int j=1;
  int kaux = k;

  if (!L) return 0;

  if (L->elem == k) return 1;

  while(kaux > L->elem)
{ kaux = kaux / 10;
          j = j*10;
          }
  if (kaux == L->elem)
     if (ricercaInizio(L->next, k % j)) return 1;
  return 0;
}
```

Esercizio 5 *Scrivere una funzione che, data un albero T di interi restituisca 1 se l'albero è simmetrico e 0 altrimenti. Ad esempio, dati i tre alberi:*



il primo è simmetrico e gli altri due no.

Soluzione Un albero è simmetrico quando il suo sottoalbero sinistro è specchiato rispetto al sottoalbero destro. Questo ci suggerisce di scomporre il problema come segue: scrivere una funzione che produce un albero specchiato, poi specchiare il sottoalbero destro e verificare che sia uguale al sottoalbero sinistro. Vediamo le tre funzioni `simmetrico`, `specchio` e `uguale`.

```
int simmetrico(tree T)
{ if (!T) return 1;
  else return uguale(T->left, specchio(T->right));
}

tree specchio(tree T)
{ tree temp;

  if (!T) return NULL;
  temp = (tree) malloc(sizeof(TreeNode));
  temp->info = T->info;
  temp->left = specchio(T->right);
  temp->right = specchio(T->left);
  return temp;
}

int uguale(tree T, tree U)
{ if (!T && !U) return 1;
else if (!T || !U) return 0;
     /* siccome la condizione !T && !U e' falsa nel
      * ramo else, se la condizione !T || !U
      * e' verificata allora UNO SOLO tra T ed U
      * e' un albero NON VUOTO
      */
else if (T->info==U->info && uguale(T->left, U->left) &&
  uguale(T->right, U->right)) return 1;
else return 0;
}
```

Osservate che la funzione `specchio` alloca nuova memoria per non distruggere l'albero di partenza. E' possibile anche scrivere direttamente una funzione analoga alla funzione

`uguale`, che controlla se un albero è il riflesso di un altro, evitando di allocare nuova memoria. In tal caso la soluzione completa potrebbe essere la seguente:

```
int simmetrico(tree T)
{ if (!T) return 1;
  else return riflesso(T->left, T->right);
}

int riflesso(tree T, tree U)
/* Osservate che fa lo stesso lavoro della funzione uguale,
 * scambiando alberi destri e sinistri
 */
{ if (!T && !U) return 1;
else if (!T || !U) return 0;
else if (T->info==U->info && riflesso(T->left, U->right) &&
  riflesso(T->right, U->left)) return 1;
else return 0;
}
```